Building an Estimator

Naruhiko Nakanishi

Abstract—The estimation portion of the controller used in the CPP simulator is developed. The simulated quad flies with the estimator and the custom controller. Each of the scenarios (Sensor Noise (scenario 6), Attitude Estimation (scenario 7), Prediction Step (scenario 8, 9), Magnetometer Update (scenario 10), Closed Loop + GPS Update (scenario 11)) is completed. An estimator is developed to be used by the controller to successfully fly a desired flight path using realistic sensors.

Index Terms—Flying Car, Quadrotor, Drone, Estimation, C++.

1 Introduction

In this project, the estimation portion of the controller used in the CPP simulator is developed. By the end of the project, the simulated quad flies with the estimator and the custom controller.

The goal of this project is as follows. An estimator is developed to be used by the controller to successfully fly a desired flight path using realistic sensors. This project is built on the same simulator as the Controls C++ project. [1]

2 Development Environment Setup

This project uses the C++ development environment. The following repository is cloned. [2]

git clone https://github.com/udacity/FCND-Estimation-CPP.git

The code is imported into the IDE like done in the Controls C++ project. The estimation simulator is compiled and run just as done in the controls project.

2.1 Project Structure

The EKF is already partially implemented in QuadEstimatorEKF.cpp. Parameters for tuning the EKF are in the parameter file QuadEstimatorEKF.txt.

When various sensors are turned on (the scenarios configure them, e.g. Quad.Sensors += SimIMU, SimMag, SimGPS), additional sensor plots become available to see what the simulated sensors measure.

The EKF implementation exposes both the estimated state and a number of additional variables.

In the config directory, in addition to finding the configuration files for the controller and the estimator, configuration files are also seen for each of the simulations.

3 The Tasks

The estimator is built up in pieces. At each step, there is a set of success criteria.

3.1 Step 1: Sensor Noise (scenario 6)

The first step to adding additional realism to the problem, and developing an estimator, is adding noise to the quad's sensors. Some simulated noisy sensor data is collected and the standard deviation of the quad's sensor is estimated.

Scenario 06 NoisySensors is chosen. In this simulation, the interest is to record some sensor data on a static quad, so the quad move is not seen.

Two plots are seen at the bottom, one for GPS X position and one for the accelerometer's x measurement. The dashed lines are a visualization of a single standard deviation from 0 for each signal.

The standard deviations are initially set to arbitrary values (after processing the data in the next step, these values are adjusted). They are set correctly, and about 68 percent of the measurement points fall into the +/- 1 sigma bound.

3.2 Step 2: Attitude Estimation (scenario 7)

In this step, the complementary filter-type attitude filter is improved with a better rate gyro attitude integration scheme.

Scenario 07 AttitudeEstimation is run. For this simulation, the only sensor used is the IMU and noise levels are set to 0. There are two plots visible in this simulation.

3.3 Step 3: Prediction Step (scenario 8, 9)

In this next step the prediction step of the filter is implemented.

3.3.1 *Scenario 08 PredictState*

Scenario 08 PredictState is run. This scenario is configured to use a perfect IMU (only an IMU). Due to the sensitivity of double-integration to attitude errors, the accelerometer is made update very insignificant (QuadEstimatorEKF.attitudeTau = 100). The plots on this simulation show element of the estimated state and that of the true state. At the moment the estimated state does not follow the true state.

3.3.2 Scenario 09 PredictionCov

A realistic IMU is introduced, one with noise. Scenario 09 PredictionCov is run. A small fleet of quadcopter is seen all using the prediction code to integrate forward.

Two plots are seen: The top graph shows 10 (prediction-only) position X estimates. The bottom graph shows 10 (prediction-only) velocity estimates however the estimated covariance (white bounds) currently do not capture the growing errors.

In QuadEstimatorEKF.cpp, the partial derivative of the body-to-global rotation matrix in the function GetRbgPrime() is calculated. Once the function is implemented, the rest of the prediction step is implemented in Predict().

3.4 Step 4: Magnetometer Update (scenario 10)

Up until now the accelerometer and gyro are only used for the state estimation. In this step, the information from the magnetometer is added to improve the filter's performance in estimating the vehicle's heading.

Scenario 10 MagUpdate is run. This scenario uses a realistic IMU, but the magnetometer update is not implemented yet. As a result, the estimate yaw is drifting away from the real value (and the estimated standard deviation is also increasing).

In this case the estimated yaw error (quad.est.e.yaw) is drifting away from zero as the simulation runs. The estimated standard deviation of that state (white boundary) is also increasing.

3.5 Step 5: Closed Loop + GPS Update (scenario 11)

Scenario 11 GPSUpdate is run. At the moment this scenario is using both an ideal estimator and an ideal IMU. Even with these ideal elements, the position and velocity errors (bottom right) are watched. They are drifting away, since GPS update is not yet implemented.

The estimator is changed by setting Quad.UseIdealEstimator to 0 in config/11 GPSUpdate.txt. The scenario is rerun to get an idea of how well the estimator work with an ideal IMU.

Realistic IMU is repeated with by commenting out these lines in config/11 GPSUpdate.txt.

3.6 Step 6: Adding the Controller (scenario 11)

Up to this point, a controller has been relaxed to work with an estimated state instead of a real state.

The controller performs well and de-tunes the controller accordingly. QuadController.cpp is replaced with the controller in the last project. QuadControlParams.txt is replaced with the control parameters in the last project.

4 Evaluation

The completion of each of the components of the estimator, and final performance of the estimator and previously made controller are evaluated.

4.1 Performance Metrics: success criteria

Performance metrics are provided for each of the different scenarios, and the controller needs to meet these minimum performance metrics for each scenario.

Scenario 6:

(step1) The standard deviations accurately capture the value of approximately 68 percent of the respective measurements.

Scenario 7:

(step2) The attitude estimator needs to get within 0.1 rad for each of the Euler angles for at least 3 seconds.

Scenario 8, Scenario 9:

(step3) This step doesn't have any specific measurable criteria being checked.

Scenario 10:

(step4) The goal is to both have an estimated standard deviation that accurately captures the error and maintain an error of less than 0.1rad in heading for at least 10 seconds of the simulation.

Scenario 11:

(step5) The objective is to complete the entire simulation cycle with estimated position error of less than 1m.

(step6) The objective is to complete the entire simulation cycle with estimated position error of less than 1m.

5 Results

QuadEstimatorEKF.cpp and QuadEstimatorEKF.txt contain the estimator and associated estimator parameters, and they successfully meet all the performance criteria. Also config/6 Sensornoise.txt is edited, and QuadController.cpp and QuadControlParams.txt contain the re-tuned controller needed to work successfully with the estimator. [3]

5.1 Step 1: Sensor Noise (scenario 6)

Fig.1 shows the result of scenario 6. The result is plugged into the top of config/6 Sensornoise.txt. Specially, the values are set for MeasuredStdDev GPSPosXY and MeasuredStdDev AccelXY to be the values which are calculated.

The simulator is run. If the values are correct, the dashed lines in the simulation eventually turn green. It indicates that approx 68 percent of the respective measurements is captured within +/- 1 sigma bound for a Gaussian noise model.

5.2 Step 2: Attitude Estimation (scenario 7)

Fig.2 shows the result of scenario 7. The top graph is showing errors in each of the estimated Euler angles. The bottom shows the true Euler angles and the estimates. Theres quite a bit of error in attitude estimation.

In QuadEstimatorEKF.cpp, the function UpdateFromIMU() contains a complementary filter-type attitude filter. To reduce the errors in the estimated attitude (Euler Angles), a better rate gyro attitude integration scheme is implemented. The attitude errors are reduced to get within 0.1 rad for each of the Euler angles.

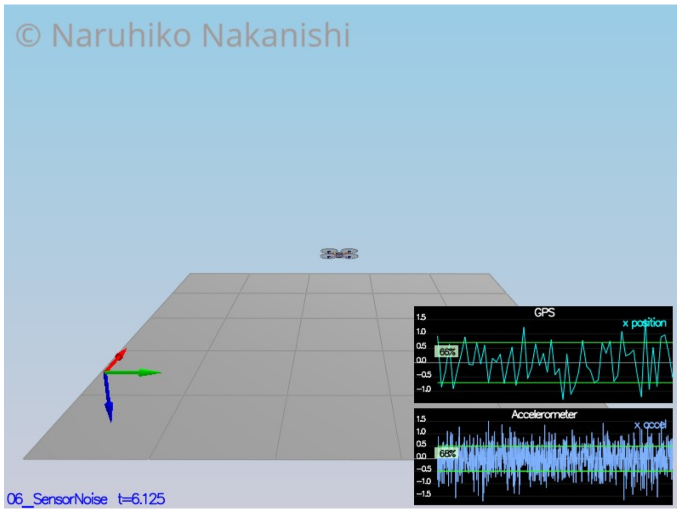

Fig. 1. scenario 6: Sensor Noise

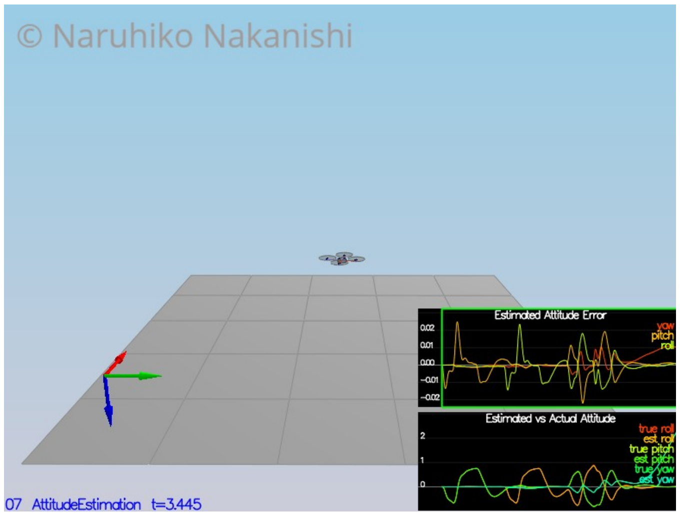

Fig. 2. scenario 7; Attitude Estimation

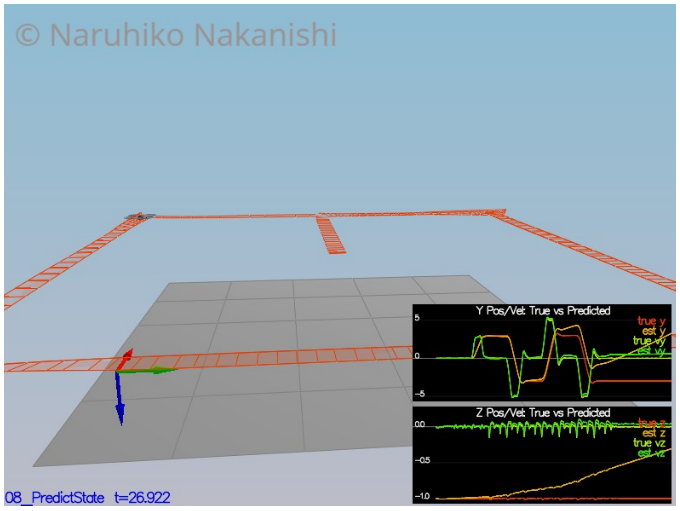

Fig. 3. scenario 8: PredictState

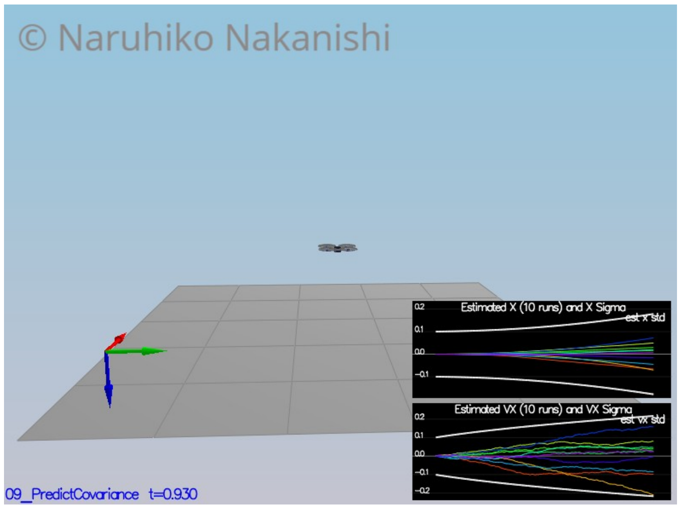
© Naruhiko Nakanishi
Estimated X (10 runs) and X Sigma
Estimated VX (10 runs) and VX Sigma
09_PredictCovariance t=0.930

Fig. 4. scenario 9: PredictionCov

5.3 Step 3: Prediction Step (scenario 8, 9)

5.3.1 Scenario 08 PredictState

Fig.3 shows the result of scenario 8. In QuadEstimatorEKF.cpp, the state prediction step is implemented in the PredictState() functon. When scenario 08 PredictState is run, the estimator state track the actual state is seen with only reasonably slow drift, as shown in the figure.

5.3.2 Scenario 09 PredictionCov

Fig.4 shows the result of scenario 9. The covariance prediction is run and the QPosXYStd and the QVelXYStd process parameters in QuadEstimatorEKF.txt are tuned to try to capture the magnitude of the error. In the first part of the plot, the covariance (the white line) grows very much like the data.

5.4 Step 4: Magnetometer Update (scenario 10)

Fig.5 shows the result of scenario 10. The parameter QYawStd (QuadEstimatorEKF.txt) for the QuadEstimatorEKF is tuned so that it approximately captures the magnitude of the drift. Magnetometer update is implemented in the function UpdateFromMag().

5.5 Step 5: Closed Loop + GPS Update (scenario 11)

Fig.6 shows the result of scenario 11. The process noise model in QuadEstimatorEKF.txt is tuned to try to approximately capture the error with the estimated uncertainty (standard deviation) of the filter. The EKF GPS Update in the function UpdateFromGPS() is implemented.

Once again the simulation is re-run. The objective is to complete the entire simulation cycle with estimated position error of less than 1m (a green box is seen over the bottom graph). The experimenting is tried with the GPS update parameters to try and get better performance.

5.6 Step 6: Adding the Controller (scenario 11)

Scenario 11 GPSUpdate is run. If the controller crashes immediately do not panic. Flying from an estimated state (even with ideal sensors) is very different from flying with ideal pose. The controller is needed to de-tune. Decrease

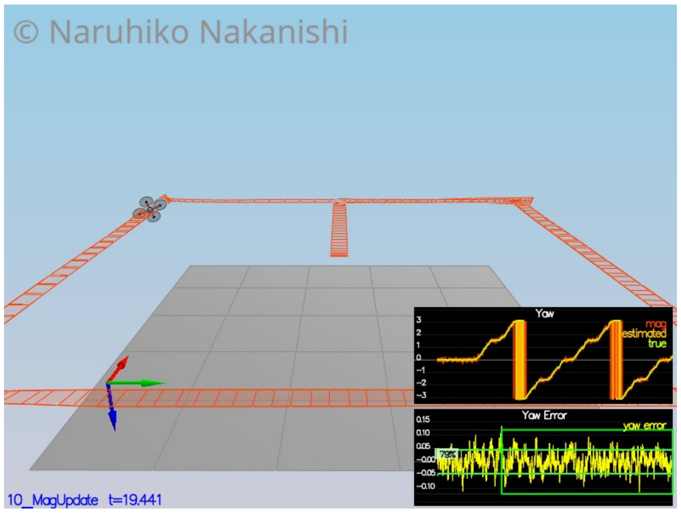

Fig. 5. scenario 10: Magnetometer Update

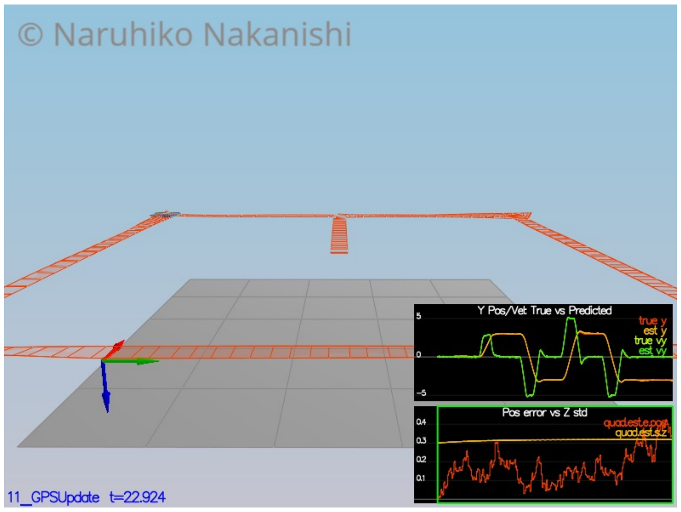

Fig. 6. scenario 11: Closed Loop + GPS Update

the position and velocity gains (weve seen about 30 percent detuning being effective) to stabilize it. The goal is to once again complete the entire simulation cycle with an estimated position error of less than 1m.

6 Conclusion

The estimation portion of the controller used in the CPP simulator is developed. The simulated quad flies with the estimator and the custom controller.

Each of the scenarios (Sensor Noise (scenario 6), Attitude Estimation (scenario 7), Prediction Step (scenario 8, 9), Magnetometer Update (scenario 10), Closed Loop + GPS Update (scenario 11)) is completed.

An estimator is developed to be used by the controller to successfully fly a desired flight path using realistic sensors.

6.1 Implement Estimator

The standard deviation of the measurement noise of both GPS X data and Accelerometer X data is determined. The calculated standard deviation correctly captures 68 percent of the sensor measurements.

A better rate gyro attitude integration scheme in the UpdateFromIMU() function is implemented. The improved integration scheme results in an attitude estimator of less than 0.1 rad for each of the Euler angles for a duration of at least 3 seconds during the simulation. The integration scheme uses quaternions to improve performance over the current simple integration scheme.

All of the elements of the prediction step for the estimator are implemented. The prediction step includes the state update element (PredictState() function), a correct calculation of the Rgb prime matrix, and a proper update of the state covariance. The acceleration is accounted for as a command in the calculation of gPrime. The covariance update follows the classic EKF update equation.

The magnetometer update is implemented. The update properly includes the magnetometer data into the state.

The GPS update is implemented. The estimator correctly incorporates the GPS information to update the current state estimate.

6.2 Flight Evaluation

The performance criteria of each step is met. For each step of the project, the final estimator successfully meets the performance criteria with the controller provided. The estimator's parameters is properly adjusted to satisfy each of the performance criteria elements.

The controller is de-tuned to successfully fly the final desired box trajectory with the estimator and realistic sensors. The controller developed in the previous project is detuned to successfully meet the performance criteria of the final scenario (less than 1m error for entire box flight).

References

[1] Udacity, "https://github.com/udacity/fcnd-controls-cpp,"
[2] Udacity, "https://github.com/udacity/fcnd-estimation-cpp,"
[3] N. Nakanishi, "https://github.com/grapestone5321/fcnd-building an estimator cpp,"